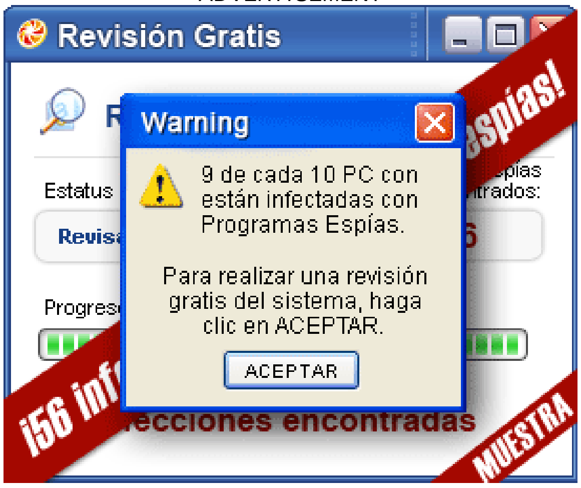
SSE2 for Dummies (who know C/C++)

by Zach Dwiel

Intro

What is SSE2?

SSE2 is an extension of assembly language which allows programs to execute one operation on multiple pieces of data at a time. Because SSE2 is assembly however, it only works on processors that support it. If the commands are attempted to be executed on a machine which is not capable of doing so, a general protection fault will be encountered. Luckily there are easy ways to tell if the processor(s) you are running on supports SSE2.


Revisión Gratis
Warning
9 de cada 10 PC con están infectadas con Programas Espías.
Para realizar una revisión gratis del sistema, haga clic en ACEPTAR.
ACEPTAR
Estatus
Progres
espías!
i56 inf
ecciones encontradas
MUESTRA

Basic Structure of SSE2

SSE2 works just like any other set of assembly calls. There are registers in which data can be stored and operations that can execute on these registers. Each register is 16 bytes (2 doubles). The 8 registers are named xmm0 through xmm7.

Basics

Some Code

```
inline void Add(double *x, double *y, double *retval)
{
  asm
  {
    // Copy the first 16 bytes into xmm0, starting at the memory x points to
    movupd xmm0, [x]
    // Copy the first 16 bytes into xmm1, starting at the memory y points to
    movupd xmm1, [y]
    // Add the 2 doubles in xmm1 to the 2 doubles in xmm0, and put the
    // result in xmm0, overwriting the previous data stored there
    addpd xmm0, xmm1
    // Copy the 16 bytes of data in xmm0 to the memory ret points to
    movupd [retval], xmm0
  }
}
```

Hopefully my comments before each line were enough to let you know what was going on. In case they weren't, I'll go into a little more detail about each line.

```
    asm{}
```

This keyword lets your compiler know that the code you are giving it will be in assembly and that it should compile it as such. It also, conveniently, tells the compiler to inline the code. This means that there is **NO** overhead for the *asm* block.

```
    movupd xmm0, [x]
    movupd xmm1, [y]
```

This command copies data from the second operand to the first; as always in Intel syntax, the asm is in dest, src order. By putting brackets around the x, we tell the mov command to copy the data that x points to the actual value of the pointer. The square brackets can be thought of as a method of dereferencing a pointer.

```
    addpd xmm0, xmm1
```

This is the line that does the actual arithmetic. It takes the value from the 2nd operand, src, and adds it to the 1st operand, dest, and stores the resulting value in the 1st operand, dest.

```
    movupd [retval], xmm0
```

Here, we copy the data that is in xmm0 to the memory that retval points to. Again, the square brackets dereference the variable retval in the same way that a '*' does in C/C++.

Some more operations

```
subpd dest, src  // subtract dest from src, store in dest
mulpd dest, src  // multiply dest and src, store in dest
divpd dest, src  // divide src by dest, store in dest
minpd dest, src  // store the smallest value, either dest or src, in dest
maxpd dest, src  // store the largest value, either dest or src, in dest
sqrtpd dest, src // take the square root of src and put the result in dest
```

A full list of SSE2 operations and a description of each can be found at HAYES Technologies.

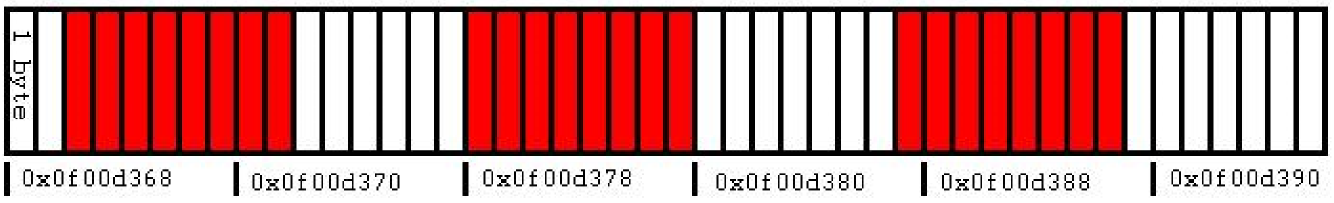
Making the Most Of SSE2

The Faster Move Instruction

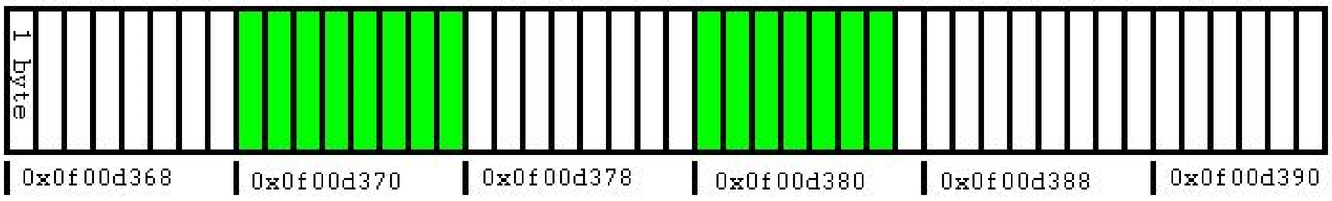
Up until now, we have been using *movapd* to move data to and from our registers. This is much slower than the instruction *movapd* which does the exact same thing, but assumes that the data is 16 byte aligned. This means that the pointer supplied must be divisible by 16. This becomes a rather large problem if you are compiling your code with gcc or the one supplied with Microsoft Visual C++. One solution to this problem is to use a different compiler such as one that Intel provides. The inherent problem with this is that the Intel compiler is not freeware like gcc. If you have already spent money on some other compiler, you probably do not want to spend more on this new compiler. One hack that I have come up with is the following:

```
#define AllignData(data) (void *)(((int)data + 15) &~ 0x0F)

//or an inline function if you prefer:

inline void *AllignData(void *data)
{
  return (void *)(((int)data + 15) &~ 0x0F);
}

void main
{
  const int sizeofdata = 512;

  double *lotsofdata;
  double *tempptr;

  tempptr = new double[sizeofdata + 2];
  lotsofdata = AllignData(tempptr);

  asm
  {
      movapd xmm0, [lotsofdata];
      // do lots of CPU intensive SSE2 operations on lotsofdata here
  }

  delete [] tempptr;
  lotsofdata = 0;
  // do not delete lotsofdata, just set it to 0, the memory it
  // points to is no longer valid
}
```

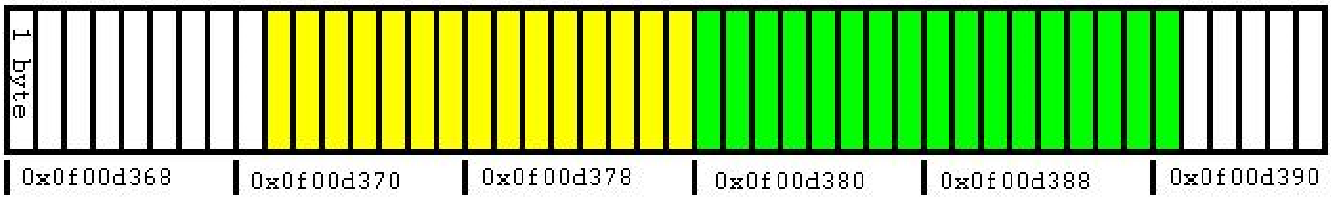
What we need for this instruction to work is memory that has a 16-byte alligned memory address.



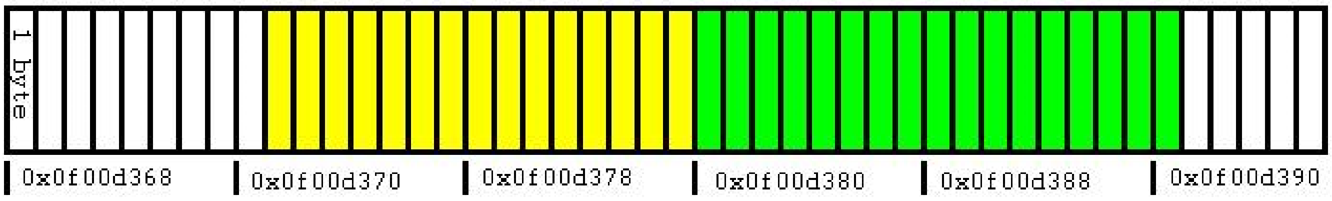
Memory that is 16byte alligned:



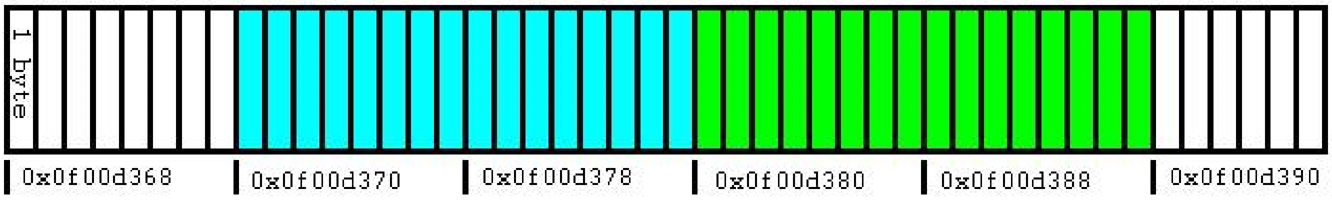
When we use the new command in C/C++ to allocate memory, we are given memory that may or may not be 16byte alligned. So, what we do, is instead of using the very first bit of our memory block, we start using it at the first place that is 16byte alligned.
-- yellow is unusbale, green is what we actually use --



By doing this, we waste the memory that comes before the first 16-byte alligned memory address. Normally this is not too big of a problem as the overhead is only once per memory block and if we are allocating large amounts of memory at once, 1 or even 12 bytes will hardly make a difference. The only problem with this is that by not using the beginning of our memory, we end up having a smaller amount of usable memory than we asked for. In the worst case, 15 bytes are not usable:



Therefore, to make sure that we get a specific number of usable bytes, we allocate at least 15 extra bytes. In the case of allocating double values, this means we must allocate an extra 2 doubles, giving us 16 extra bytes:



If you are not using doubles, just make sure that at least 15 extra bytes are allocated. Just determine the size of the data type and compute how many are needed to give you the needed padding.

Another important thing to notice is that we delete the variable that holds the original pointer to all of our memory. If you try to delete the memory starting in the middle of our memory block, you cause a general protection fault and have a memory leak.

Ordering Your Operations

One thing that is often over looked is the order that registers are used. When an operation is performed, there is a delay while the result is bieng moved to its destination. if the next operation requires this value, it must wait for it to be stored into the register. If however, the next operation does not need this data, it does not need to wait for it to be stored, it can go ahead and execute at the same time that the previous result is getting stored.

For instance, there will be a speed difference between the following two code segments:

#1:

```
asm
{
  movupd xmm0, [x]   // xmm0 = x
  movupd xmm1, [y]   // xmm1 = y
  movupd xmm2, [z]   // xmm2 = z
  movupd xmm3, [w]   // xmm3 = w

  movapd xmm4, xmm2  // xmm4 = z
  movapd xmm5, xmm1  // xmm5 = y
  movapd xmm6, xmm0  // xmm6 = x

  addpd xmm2, xmm3   // xmm2 = z + w
  addpd xmm1, xmm2   // xmm1 = y + z + w
  addpd xmm0, xmm1   // xmm0 = x + y + z + w

  mulpd xmm4, xmm3   // xmm4 = z * w
  mulpd xmm5, xmm4   // xmm5 = y * z * w
  mulpd xmm6, xmm5   // xmm6 = x * y * z * w

  divpd xmm0, xmm6   // xmm0 = (x * y * z * w) / (x + y + z + w)

  movupd [ret], xmm0 // ret = (x * y * z * w) / (x + y + z + w)
}
```

#2:

```
asm
{
  movupd xmm0, [x]   // xmm0 = x
  movupd xmm1, [y]   // xmm1 = y
  movupd xmm2, [z]   // xmm2 = z
  movupd xmm3, [w]   // xmm3 = w

  movapd xmm4, xmm2  // xmm4 = z
  movapd xmm5, xmm1  // xmm5 = y
  movapd xmm6, xmm0  // xmm6 = x

  addpd xmm2, xmm3   // xmm2 = z + w
  mulpd xmm4, xmm3   // xmm4 = z * w
  addpd xmm1, xmm2   // xmm1 = y + z + w
  mulpd xmm5, xmm4   // xmm5 = y * z * w
  addpd xmm0, xmm1   // xmm0 = x + y + z + w
  mulpd xmm6, xmm5   // xmm6 = x * y * z * w

  divpd xmm0, xmm6   // xmm0 = (x * y * z * w) / (x + y + z + w)

  movupd [ret], xmm0 // ret = (x * y * z * w) / (x + y + z + w)
}
```

The second piece of code will run faster. This is because in the second case, there are only 2 cases where one instruction relies on the data from the previous one to perform its computations. Because of this, instructions can be executed immediately after the previous one finsihes instead of waiting for it to store its result in the registers.

Don't get carried away

A common mistake made by people new to SSE2 is to convert a lot of their old and future code into SSE2. This can actually result in slower code. The reason for this is the very large overhead for the CPU to copy memory to the registers. If you have an application that is doing a small number of operations on a large data set, you can expect to be less efficient than if you are doing a lot of operations on a small amount of data.

Compiling SSE2 with gcc/g++

The first thing that you need to remember to do when you want to compile SSE2 embedded C/C++ code with gcc/g++, is to throw in the -masm=intel switch during compile. You must also put ".intel_syntax noprefix" in front of your asm code and surround it with quotes like this:

```
    asm(".intel_syntax noprefix\n");
    asm("    mov eax, x\n");
    asm("    movupd xmm0, [eax+0x00]\n");
    asm("    movupd xmm1, [eax+0x10]\n");
    asm("    addpd xmm0, xmm1\n");
    asm("    movupd [eax+0x20], xmm0\n");

    or

    asm(".intel_syntax noprefix
         mov eax, x
         movupd xmm0, [eax+0x00]
         movupd xmm1, [eax+0x10]
         addpd xmm0, xmm1
         movupd [eax+0x20], xmm0\n");
```

Note that the asm block is inside "()" not "{}". Also, if you want to use a variable declared in your C/C++ code, you must define it **publicly**. Any variables defined locally, whether inside your main function, inside a for loop, etc, will not be seen by the linker and will be considered an "undefined reference".

End

Questions? Comments? Suggestions? mis-spellings? Grammatical Errors? Email me at dwiel@insightbb.com

Author: Zach Dwiel
Last Updated: 08/23/03

Discuss this article in the forums

Date this article was posted to GameDev.net: **8/31/2003**
(Note that this date does not necessarily correspond to the date the article was written)

See Also:
All
x86 Assembly